XT5 Compiler Resources

Outline

Foreword

- Source code that is readable, maintainable, and produces correct results is not always organized for efficient execution. Normally, the first step in the program development process involves producing code that executes and produces the correct results. This first step usually involves compiling without much worry about optimization. After code is compiled and debugged, code optimization and parallelization become an issue.

- Invoking one of the PGI/GNU/Intel/Cray/Pathscale compiler commands with certain options instructs the compiler to generate optimized code. Optimization is not always performed since it increases compilation time and may make debugging difficult. However, optimization produces more efficient code that usually runs significantly faster than code that is not optimized.

Outline: Introduction

Parallel Compiling on Jaguar

- Jaguar has two kinds of nodes:
  - Compute Nodes running the CNL OS
  - Service and login nodes running Linux
- To build a code for the compute nodes, you should use the Cray wrappers `cc`, `CC`, and `ftn`. The wrappers will call the appropriate compiler which will use the appropriate header files and link against the appropriate libraries. Use of wrappers is crucial for building the parallel codes on Cray.
- We highly recommend that the `cc`, `CC`, and `ftn` wrappers be used when building for the compute nodes! Both parallel and serial codes.
- To build a code for the Linux service nodes, you should call the compilers directly.
- We strongly suggest that you don't call the compilers directly if you are building code to run on the compute nodes.

System Parallel Compilers

Language	Compiler
C	`cc`
C++	`CC`
Fortran 77, 90 and 95	`ftn`

Wrappers and Compiling Tips

- Why to use wrappers to build (compile and link) the code:
  - Automatically point to correct compiler based on modules loaded
  - Wrappers automatically find and include paths and libraries of loaded modules (e.g., mpi, libsci)

- Use same makefile for all compilers*

- Calling base compilers directly (e.g., `pgf90`) results in serial code that runs only on login nodes
  - Not what you want! Use wrapper instead and run on compute nodes
  - Discourteous to other users to do production work on login nodes

System Serial Compilers

- Available compilers:
  - Portland Croup (PGI). Module name: PrgEnv-pgi
    - `pgcc`
    - `pgCC`
    - `pgf90/pgf95`
    - `pgf77`
  - GNU. Module name: PrgEnv-gnu
    - `gcc`
    - `g++`
    - `Gfortran`
  - Intel. Module name: PrgEnv-intel
    - `icc` (c/c++ codes)
    - `ifort`
  - Cray compilers. Module name: PrgEnv-cray
    - `craycc`
    - `crayCC`
    - `crayftn`
  - Pathscale. Module name: PrgEnv-pathscale
    - `pathcc`
    - `pathCC`
    - `path90/pathf95` (only available if gcc/4.2.1 or higher is loaded)

Note that the `man` pages for the system compilers will only give the most basic information, i.e.

```
%man cc
%man CC
%man ftn
```

The `man` pages with the specific compiler options can be accessed by using the names of the serial compilers on this slide:

```
%man pgcc
%man g++
%man crayftn
```

Default Compilers

- Default compiler is PGI. The list of all packages is obtained by
  - `module avail PrgEnv`
- To use the Cray wrappers with other compilers the programming environment modules need to be swapped, i.e.
  - `module swap PrgEnv-pgi PrgEnv-gnu`
  - `module swap PrgEnv-pgi PrgEnv-cray`
  - `module swap PrgEnv-pgi PrgEnv-intel`
  - `module swap PrgEnv-pgi PrgEnv-pathscale`
- To just use the GNU/Cray compilers directly load the GNU/Cray module you want:
  - `module load PrgEnv-gnu/2.1.50HD`
  - `module load PrgEnv-cray/1.0.1`

MPI Codes

- *All system compilers (PGI, GNU, Intel, Cray, Pathscale) can handle MPI standard specification parallel codes through the use of compiler wrappers (cc, ftn, CC)*
- MPT – Cray's MPI library
  - Use latest MPT (4.0.0)
- Default settings are
  - Some codes may benefit from setting or adjusting the environment variable settings.
- More information is available on man pages "man mpi"

Outline: PGI® compilers

- Portland Group (PGI)
- List of the Compiler Option Categories
- PGI Basic Compiler Usage
- Flags to support language dialects
- Specifying the target architecture
- Flags for debugging aids
- Useful Compiler Flags

Portland Group (PGI)

- Cray provides the Portland Group (PGI) compilers as part of several programming environments on Jaguar.
- PGI compilers are loaded by default.

List of the Compiler Option Categories

- Overall Options
- Optimization Options (covered in this document)
- Debugging Options
- Preprocessor Options
- Assembler Options
- Linker Options
- Language Options
- Target-specific Options

PGI Basic Compiler Usage

- A compiler driver interprets options and invokes pre-processors, compilers, assembler, linker, etc.
- Options precedence: if options conflict, last option on command line takes precedence
- Use -Minfo and -Mneginfo to see a listing of optimizations and transformations performed by the compiler
- Use -help to list all options or see details on how to use a given option, e.g. pgf90 -Mvect -help
- Use man pages for more details on options, e.g. “man pgf90”
- Use –v to see under the hood

Flags to support language dialects

- Fortran
  - ftn
  - Suffixes .f, .F, .for, .fpp, .f90, .F90, .f95, .F95
  - -Mextend, -Mfixed, -Mfreeform
  - Type size –i2, -i4, -i8, -r4, -r8, etc.
  - -Mcray, -Mbyteswapio, -Mupcase, -Mnomain, -Mrecursive, etc.
- C/C++
  - cc, CC
  - Suffixes .c, .C, .cc, .cpp, .i
  - -B, -c89, -c9x, -Xa, -Xc, -Xs, -Xt
  - -Msignextend, -Mfcon, -Msingle, -Muchar, -Mgccbugs

Specifying the target architecture

- `-tp target` - Specify the type of the target processor;
- The default in the absence of the -tp flag is to compile for the type of CPU on which the compiler is running.
- The targets are:
  - `-tp k8-64` - AMD Opteron or Athlon-64 in 64-bit mode.
  - `-tp amd64e` - AMD Opteron revision E or later, in 64-bit mode; includes SSE3 instructions
  - `-tp x64` - Single binary where each procedure is optimized for the AMD Opteron in 64-bit mode; the selection of which optimized copy to execute is made at run time depending on the machine executing the code.
  - `-tp k8-32,k7,p7,piv,piii,p6,p5,px` for 32 bit code

Flags for debugging aids

- -g generates symbolic debug information used by a debugger
- -gopt generates debug information in the presence of optimization
- -Mbounds adds array bounds checking
- -v gives verbose output, useful for debugging system or build problems
- -Minfo provides feedback on optimizations made by the compiler
- -S or –Mkeepasm to see the exact assembly generated

Useful Compiler Flags

General

Flag	Comments
`-mp=nonuma`	Compile multithreaded code using OpenMP directives

Debugging

Flag	Comments
`-g`	For debugging symbols; put first
`-Ktrap=fp`	Trap floating point exceptions
`-Mchkptr`	Checks for unintended dereferencing of null pointers

Optimization-Related PGI Compiler Options

- The compilers optimize code according to the specified optimization level. You can use a number of options to specify the optimization levels, including –O, –Mvect, –Mipa, and –Mconcur. In addition, you can use several of the –M<pgflag> switches to control specific types of optimization and parallelization.
- The optimization options are:

–fast	–Minline	–Mpfi	–Mvect
–Mconcur	–Mipa=fast	–Mpfo	–O
–Minfo	–Mneginfo	–Munroll	–Msafeptr

Optimization-Related PGI Compiler Options (continued)

Option	Description
`-fast`	Generally optimal set of flags for targets that support SSE capability.
`-fastsse`	Generally optimal set of flags for targets that include SSE/SSE2 capability.
`-M<pgflag>`	Selects variations for code generation and optimization.
`-mp[=all, align, bind, [no]numa]`	Interpret and process user-inserted shared-memory parallel programming directives.
`-O<level>`	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
`-pc <val>`	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
`-Mprof=time`	Instrument the generated executable to produce a gprof-style

Optimization-Related PGI Compiler Options (continued)

- Traditional optimization controlled through -O[<n>], n is 0 to 4.
- -fastsse and -fast are equal to -O2 -Munroll=c:1 -Mnoframe –Mlre -Mvect=sse, -Mscalarsse -Mcache_align -Mflushz
  - For -Munroll, c specifies completely unroll loops with this loop count or less
  - -Munroll=n:<m> says unroll other loops m times
- -Mcache_align aligns top level arrays and objects on cache-line boundaries
- -Mflushz flushes SSE denormal numbers to zero
- -Mnoframe does not set up a stack frame
- -Mlre is loop-carried redundancy elimination

Outline: Getting Started with PGI Compiler Optimizations

Quick Start

- To get started quickly with optimization, a good set of options to use with any of the PGI compilers is –fast –Mipa=fast. For example:

  ```
  $ ftn -fast -Mipa=fast prog.f
  ```

- For all of the PGI Fortran, C, and C++ compilers, the –fast –Mipa=fast options generally produce code that is well-optimized without the possibility of significant slowdowns due to pathological cases.
  - The –fast option is an aggregate option that includes a number of individual PGI compiler options; which PGI compiler options are included depends on the target for which compilation is performed.
  - The –Mipa=fast option invokes interprocedural analysis including several IPA suboptions.
  - For C++ programs, add -Minline=levels:10 --no_exceptions as shown here:

    ```
    $ CC -fast -Mipa=fast -Minline=levels:10 --no_exceptions prog.cc
    ```

-help and -Minfo

–help

- You can see a specification of any command line option by invoking any of the PGI compilers with -help in combination with the option in question, without specifying any input files. For example, you might want information on -O:
- $ pgf95 -help –O
- Or you can see the full functionality of -help itself, which can return information on either an individual option or groups of options:
- $ pgf95 -help –help

–Minfo

- Used to display compile-time optimization listings.
- When this option is used, the PGI compilers issue informational messages to stderr as compilation proceeds. From these messages, you can determine which loops are
  - optimized using unrolling,
  - SSE instructions,
  - vectorization,
  - parallelization,
  - interprocedural optimizations
  - various miscellaneous optimizations.
- you can also see where and whether functions are inlined.

–Mneginfo and –dryrun

–Mneginfo
- Used to display informational messages listing why certain optimizations are inhibited.

–dryrun
- Can be useful as a diagnostic tool if you need to see the steps used by the compiler driver to preprocess, compile, assemble and link in the presence of a given set of command line inputs.
- If –dryrun option is specified, these steps will be printed to stderr but are not actually performed.
- For example, you can use this option to inspect the default and user-specified libraries that are searched during the link phase, and the order in which they are searched by the linker.

Common Performance Challenges

- **Vectorization**
  - What is vectorization? Is my code vectorizing?
  - Conflicts with C++ and F90 “ease of use” programming techniques. C and C++ pointer issues that prevent vectorization.
- **Multi-core issues**
  - Memory bandwidth
  - MPI, OpenMP, and auto parallelization
- **IPA** – Interproceedural Analysis and Inlining
  - IPA and inline enabled libraries

What is Vectorization on x64 CPUs?

- **By a Programmer**: writing or modifying algorithms and loops to enable or maximize generation of x64 packed Streaming SIMD Extensions (SSE) instructions by a vectorizing compiler
- **By a Compiler**: identifying and transforming loops to use packed SSE arithmetic instructions which operate on more than one data element per instruction
- For more information, please, refer to the "*Software Optimization Guide for AMD64 Processors*" at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF

Optimization Strategies

- Establish a workload
- Optimization from the top-down
- Use of proper tools, methods
- Processor level optimizations, parallel methods
- Different flags/features for different types of code

Outline: Optimization Categories (Node Level Tuning)

- Local and Global Optimization
- Vectorization
- Interprocedural Analysis (IPA)
- Function Inlining
- SMP Parallelization
- Miscellaneous Optimizations

Local and Global Optimization

Local Optimization

- This optimization is performed on a block-by-block basis within a program's basic blocks. A basic block is a sequence of statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching, except at the end.
- The PGI compilers perform many types of local optimization including:
  - algebraic identity removal,
  - constant folding,
  - common sub-expression elimination,
  - redundant load and store elimination,
  - scheduling,
  - strength reduction,
  - peephole optimizations.

Global Optimization

- This optimization is performed on a program unit over all its basic blocks. The optimizer performs control-flow and data-flow analysis for an entire program unit. All loops, including those formed by IFs and GOTOs, are detected and optimized.
- Global optimization includes:
  - constant propagation,
  - copy propagation,
  - dead store elimination,
  - global register allocation,
  - invariant code motion,
  - induction variable elimination.

Local and Global Optimization using –O

Using the PGI compiler commands with the –Olevel option (the capital O is for Optimize), you can specify any of the following optimization levels:

–O0 Level zero specifies no optimization. A basic block is generated for each language statement.

–O1 Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.

–O2 Level two specifies global optimization. This level performs all level-one local optimization as well as level two global optimization. If optimization is specified on the command line without a level, level 2 is the default.

–O3 Level three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

–O4 Level four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

Note: If you use the –O option to specify optimization and do not specify a level, then level-two optimization (–O2) is the default.

Menu

Local and Global Optimization using –O (continued)

- You can explicitly select the optimization level on the command line. For example, the following command line specifies level-two optimization which results in global optimization:
- $ pgf95 -O2 prog.f
- Specifying –O on the command-line without a level designation is equivalent to –O2. The default optimization level changes depending on which options you select on the command line. For example, when you select the –g debugging option, the default optimization level is set to level-zero (–O0). However, if you need to debug optimized code, you can use the -gopt option to generate debug information without perturbing optimization.
- As noted previously, the –fast option includes –O2 on all x86 and x64 targets. If you want to override the default for–fast with –O3 while maintaining all other elements of –fast, simply compile as follows:
- $ pgf95 -fast -O3 prog.f

Vectorization

Loop Optimization: Unrolling, Vectorization, and Parallelization

- The performance of certain classes of loops may be improved through vectorization or unrolling options.
  - Vectorization transforms loops to improve memory access performance and make use of packed SSE instructions which perform the same operation on multiple data items concurrently.
  - Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions.
- Performance for loops on systems with multiple processors may also improve using the parallelization features of the PGI compilers.

Vectorizable F90 Array Syntax Data is REAL*4


```
350 !
351 !   Initialize vertex, similarity and coordinate arrays
352 !
353     Do Index = 1, NodeCount
354       IX = MOD (Index - 1, NodesX) + 1
355       IY = ((Index - 1) / NodesX) + 1
356       CoordX (IX, IY) = Position (1) + (IX - 1) * StepX
357       CoordY (IX, IY) = Position (2) + (IY - 1) * StepY
358       JetSim (Index)  = SUM (Graph (:, :, Index) * &
359     &                       GaborTrafo (:, :, CoordX(IX,IY), CoordY(IX,IY)))
360       VertexX (Index) = MOD (Params%Graph%RandomIndex (Index) - 1, NodesX) + 1
361       VertexY (Index) = ((Params%Graph%RandomIndex (Index) - 1) / NodesX) + 1
362     End Do
```

Inner “loop” at line 358 is vectorizable, can used packed SSE instructions

-fastsse -Minfo

```
% pgf95 -fastsse -Mipa=fast -Minfo -S graphRoutines.f90
…
localmove:
   334, Loop unrolled 1 times (completely unrolled)
   343, Loop unrolled 2 times (completely unrolled)
   358, Generated an alternate loop for the inner loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
```

Scalar SSE vs. Vector SSE

Scalar SSE:

```
 LB6_668:
# lineno: 358
        movss   -12(%rax),%xmm2
        movss   -4(%rax),%xmm3
        subl    $1,%edx
        mulss   -12(%rcx),%xmm2
        addss   %xmm0,%xmm2
        mulss   -4(%rcx),%xmm3
        movss   -8(%rax),%xmm0
        mulss   -8(%rcx),%xmm0
        addss   %xmm0,%xmm2
        movss   (%rax),%xmm0
        addq    $16,%rax
        addss   %xmm3,%xmm2
        mulss   (%rcx),%xmm0
        addq    $16,%rcx
        testl   %edx,%edx
        addss   %xmm0,%xmm2
        movaps  %xmm2,%xmm0
        jg      .LB6_625
```

Vector SSE:

```
 LB6_1245:
# lineno: 358
        movlps  (%rdx,%rcx),%xmm2
        subl    $8,%eax
        movlps  16(%rcx,%rdx),%xmm3
        prefetcht0  64(%rcx,%rsi)
        prefetcht0  64(%rcx,%rdx)
        movhps  8(%rcx,%rdx),%xmm2
        mulps   (%rsi,%rcx),%xmm2
        movhps  24(%rcx,%rdx),%xmm3
        addps   %xmm2,%xmm0
        mulps   16(%rcx,%rsi),%xmm3
        addq    $32,%rcx
        testl   %eax,%eax
        addps   %xmm3,%xmm0
        jg      .LB6_1245:
```

Facerec Scalar: 104.2 sec

Facerec Vector: 84.3 sec

Vectorizable C Code Fragment?


```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222         u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224         float vdt = v[i] * dt;
225         u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Minfo functions.c
func4:
    221, Loop unrolled 4 times
    221, Loop not vectorized due to data dependency
    223, Loop not vectorized due to data dependency
```


Menu

Pointer Arguments Inhibit Vectorization

```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222       u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224       float vdt = v[i] * dt;
225       u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Msafeptr –Minfo functions.c
func4:
    221, Generated vector SSE code for inner loop
         Generated 3 prefetch instructions for this loop
    223, Unrolled inner loop 4 times
```


Menu

C Constant Inhibits Vectorization


```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222       u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224       float vdt = v[i] * dt;
225       u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Msafeptr –Mfcon –Minfo functions.c
func4:
    221, Generated vector SSE code for inner loop
         Generated 3 prefetch instructions for this loop
    223, Generated vector SSE code for inner loop
         Generated 4 prefetch instructions for this loop
```


Menu

-Msafeptr Option and Pragma

–M[no]safeptr[=all | arg | auto | dummy | local | static | global]

all	all pointers are safe
arg	argument pointers are safe
local	local pointers are safe
static	static local pointers are safe
global	global pointers are safe

Common Barriers to SSE Vectorization

Potential Dependencies & C Pointers – Give compiler more info with –Msafeptr, pragmas, or restrict type qualifer

Function Calls – Try inlining with –Minline or –Mipa=inline

Type conversions – manually convert constants or use flags

Too few iterations – Usually better to unroll the loop

Real dependencies – Must restructure loop, if possible

Barriers to Efficient Execution of Vector SSE Loops

- Not enough work – vectors are too short
- Vectors not aligned to a cache line boundary
- Non-unity strides
- May run out of space to handle all the instructions
- **Code bloat if altcode is generated**

Interprocedural Analysis (IPA)

Interprocedural Analysis (IPA) and Optimization

- Interprocedural analysis (IPA) allows use of information across function call boundaries to perform optimizations that would otherwise be unavailable. For example, if the actual argument to a function is in fact a constant in the caller, it may be possible to propagate that constant into the callee and perform optimizations that are not valid if the dummy argument is treated as a variable.
- A wide range of optimizations are enabled or improved by using IPA, including but not limited to
  - data alignment optimizations,
  - argument removal,
  - constant propagation,
  - pointer disambiguation,
  - pure function detection,
  - F90/F95 array shape propagation,
  - data placement,
  - vestigial function removal,
  - automatic function inlining,
  - inlining of functions from pre-compiled libraries,
  - interprocedural optimization of functions from pre-compiled libraries.

What can Interprocedural Analysis and Optimization with –Mipa do for You?

- Interprocedural constant propagation
- Pointer disambiguation
- Alignment detection, Alignment propagation
- Global variable mod/ref detection
- F90 shape propagation
- Function inlining

Effect of IPA on the WUPWISE Benchmark

PGF95 Compiler Options	Execution Time in Seconds
–fastsse	156.49
–fastsse –Mipa=fast	121.65
–fastsse –Mipa=fast,inline	91.72

- –Mipa=fast => constant propagation => compiler sees complex matrices are all 4x3 => completely unrolls loops

- –Mipa=fast,inline => small matrix multiplies are all inlined

Using Interprocedural Analysis

- Must be used at both compile time and link time
- Non-disruptive to development process – edit/build/run
- Speed-ups of 5% - 10% are common
- –Mipa=safe:<*name*> - safe to optimize functions which call or are called from unknown function/library *name*
- –Mipa=libopt – perform IPA optimizations on libraries
- –Mipa=libinline – perform IPA inlining from libraries

Function Inlining

- This optimization allows a call to a function to be replaced by a copy of the body of that function. This optimization will sometimes speed up execution by eliminating the function call and return overhead.
- Function inlining may also create opportunities for other types of optimization.
- Function inlining is not always beneficial.
- When used improperly it may increase code size and generate less efficient code.

Explicit Function Inlining

–Minline[=[lib:]<inlib> | [name:]<func> | except:<func> | size:<n> | levels:<n>]

[lib:]<inlib>	Inline extracted functions from *inlib*
[name:]<func>	Inline function func
except:<func>	Do not inline function func
size:<n>	Inline only functions smaller than n statements (approximate)
levels:<n>	Inline n levels of functions

Specific recommendations for C++

- Encapsulation; Data hiding
  - small functions, inline!
- Exception handling
  - use `-no_exceptions` until 7.0
- Overloaded operators, overloaded functions
  - Can be used
- Pointer Chasing
  - `-Msafeptr`, restrict qualifier
- Templates, Generic Programming
  - Can be used. However, aggressive use of templates may still run into problems
- Inheritance, polymorphism, virtual functions
  - runtime lookup or check, no inlining, potential performance penalties

Miscellaneous Optimizations

- –Mfprelaxed
  - single-precision sqrt, rsqrt, div performed using reduced-precision reciprocal approximation.
  - *Caution: This should only be used if the code can tolerate a loss of precision (2-3 decimal points)*
- –Mprefetch=d:<p>,n:<q>
  - control prefetching distance, max number of prefetch instructions per loop
- –M[no]movnt
  - disable / force non-temporal moves
- –V[version]
  - to switch between PGI releases at file level
- –Mvect=noaltcode
  - disable multiple versions of loops

PGI Documentation and Support

- The Portland Group website
  - http://www.pgroup.com/
- PGI provided documentation
  - http://www.pgroup.com/resources/docs.htm
- PGI User Forums
  - https://www.pgroup.com/userforum/index.php
- PGI FAQs, Tips & Techniques pages

Cray X86 compilers

- Cray provides its own Cray X86 high-performance compiler set as part of several programming environments on Jaguar.

- To switch to Cray X86 compilers from PGI compilers loaded by default, please, refer to the Introduction section of this document.

Outline: Getting Started with Cray Compiler Optimizations

- Quick Start
- Directives
- Current Strengths

Quick Start

- Make sure it is available
  - module avail PrgEnv-cray
- To access the Cray compiler
  - module load PrgEnv-cray
- To target the various chips
  - module load xtpe-barcelona,shanghi,istanbul]
- Once you have loaded the module “cc” and “ftn” are the Cray compilers
  - Recommend just using default options
  - Use –rm (fortran) and –hlist=m (C) to find out what happened
- Example: ftn –rm –c file.f90

Resources for Users: Optimization-Related References

- *Software Optimization Guide for AMD64 Processors* (Guidelines for serial optimizations specific to AMD Opteron on the AMD site):
  http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF
- OpenMP Specifications/Documentation:
  http://openmp.org/wp/openmp-specifications/
- OpenMP tutorial from LLNL
  https://computing.llnl.gov/tutorials/openMP/

Resources for Users: Getting Started

- PGI Compilers for XT5
  - http://www.pgroup.com/resources/docs.htm
- Cray Compilers
  - http://docs.cray.com/
- Gnu Compilers
  - http://gcc.gnu.org/onlinedocs/
- Intel Compilers
  - http://software.intel.com/en-us/intel-compilers/
- Pathscale Compilers
  - http://www.pathscale.com/

Resources for Users: More Information

- NCCS website

  http://www.nccs.gov/

- Cray Documentation

  http://docs.cray.com/

- Contact us

  help@nccs.gov